

StarShow往期精华01：向量化编程的精髓 - 康凯森

数据库软硬件技术的发展，带来了对 CPU 性能的极限压榨，提升数据库计算引擎性能成了关注的焦点。传统的火山模型效率偏低，业界探索、实践总结出了向量化编程的方法，有效地提升了数据库系统的性能。本文将具体介绍 StarRocks 在向量化编程方面的相关探索、实践以及一些前沿的思考。

01：向量化基础知识

在深入向量化编程之前，先简单回顾一下 CPU 指令执行的背景知识。一般而言，CPU 指令执行包含以下 5 个步骤：

1. 取指令
2. 指令译码
3. 执行指令
4. 访存取数
5. 结果写回

优化 CPU 的性能是向量化编程的核心，其公式可归纳为 $\text{CPU Time} = \text{Instruction Number} \times \text{CPI} \times \text{Clock Cycle Time}$ ，其中前两项是优化的核心。具体到优化分析，可以用图 1 所示的分析方法，重点聚焦于 Retiring、Bad Speculation、Frontend Bound 和 Backend Bound 等 4 个方面。

CPU Performance Analysis Top-Down Hierarchy

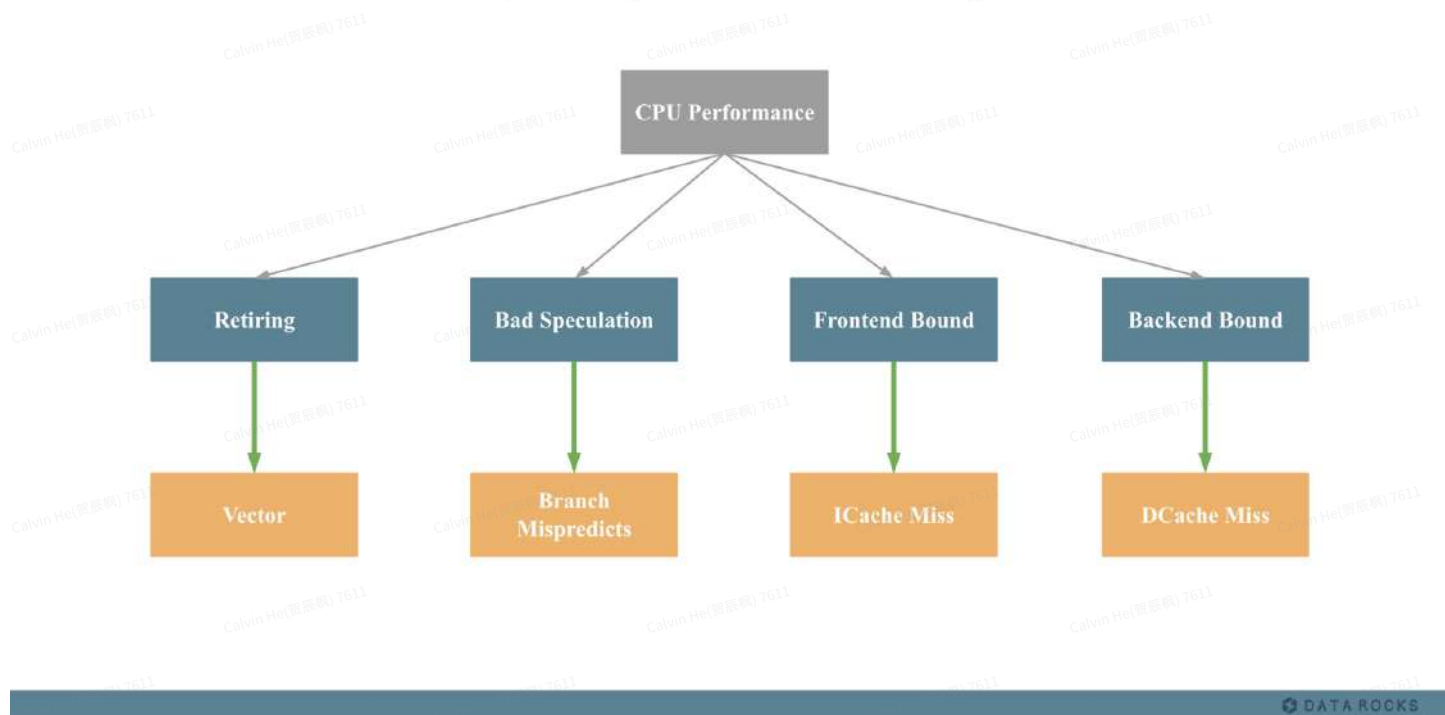


图 1

SIMD 指令

SIMD(Single Instruction Multiple Data)全称单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。

如图 2 所示，相较于传统的 SISD（单指令单数据流）指令，SIMD 指令最大的特点是具有多个元素相同操作只需使用一条指令执行，由此带来了性能的大幅提升，最高可达 16 倍。

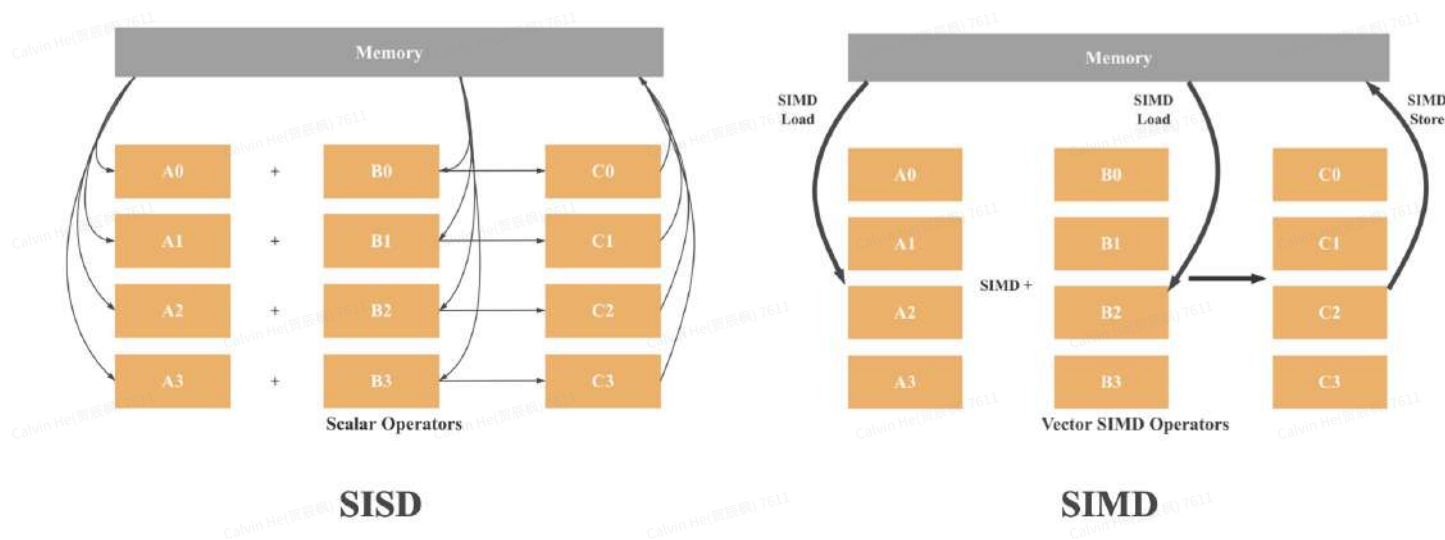


图 2

为支持向量化编程，数据库引入了 SIMD 寄存器，从最初的 128 位到最新的 512 位，位宽一直在持续加大，可以处理的数据量也变得越来越来多。

向量化编程的 6 种方式

一般而言，向量化编程有 6 种方式，如图 3 所示。在常规的数据库开发过程中，触发编译器的自动向量化是理想化状态，手动向量化操作处理对研发人员的要求相对较高，6 种方式总结下来包括：

- Compiler: Auto-vectorization
- Compiler: Auto-vectorization hints
- Compiler: OpenMP* 4.0 以及 Intel® Clik™ Plus
- SIMD intrinsic class
- Vector intrinsic
- Assembler code

Many Ways to Vectorize

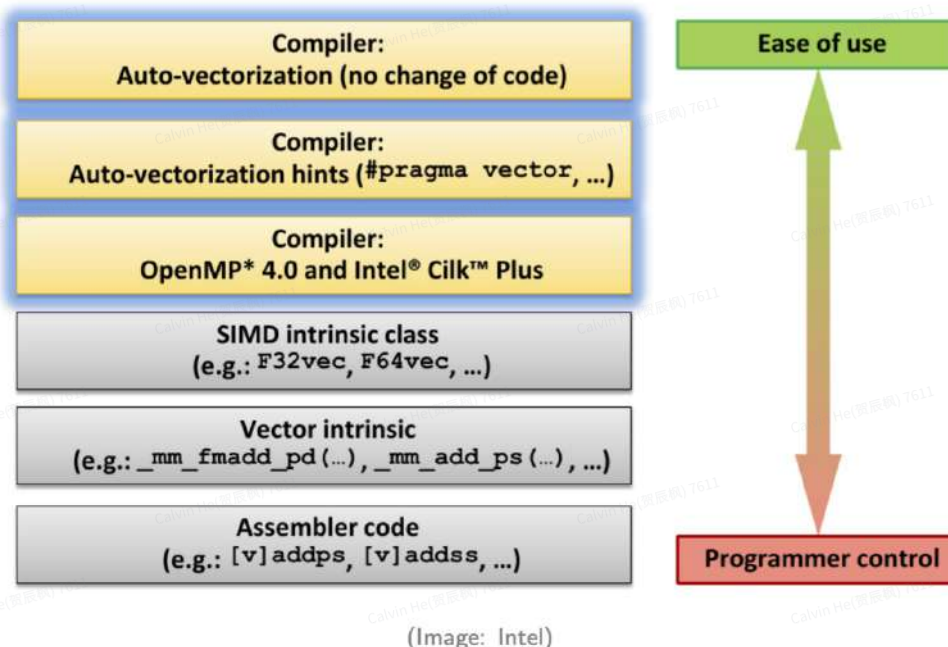


图 3

编译器自动向量化

编译器本身一直处于动态的发展过程中，编译器自动向量化的触发条件相对比较苛刻，基本上要满足：循环次数是确定的、函数调用应该是内联的 or 简单的数学函数、无数据依赖、循环内无复杂条件等 4 点。

我们可以通过给编译器加 Restrict 关键词，用 Hint 的方式帮助编译器理解代码，触发自动向量化。如图 4 所示，两组代码的区别在于第二组代码加上了 Restrict 关键词，编译器接收到了相关信息，进而可以自动优化，经过 Benchmark 实测，第二组代码的性能是第一组的数倍。

Compile Hint Vectorize: Restrict

```
void batch_update(int* res, int* col, int size) {  
    for (int i = 0; i < size; ++i) {  
        *res += col[i];  
    }  
}
```

Which is Faster ?

```
void batch_update_restrict(int* __restrict res, int* __restrict col,  
    for (int i = 0; i < size; ++i) {  
        *res += col[i];  
    }  
}
```

DATA ROCKS

图 4

如何判断代码是否被编译器向量化，一般有以下两种方式。

第一种，在编译时加参数：

- -fopt-info-vec-optimized
- -fopt-info-vec-missed
- -fopt-info-vec-note
- -fopt-info-vec-all

第二种相对直接，可以利用一些在线网站直接看其生成的汇编代码，如图 5 所示，如果右侧汇编代码中出现有 xmm、ymm、zmm 等指令，则代表编译器已进行 SIMD 指令的优化，自动向量化生效。

How to Ensure SIMD Instructions Used

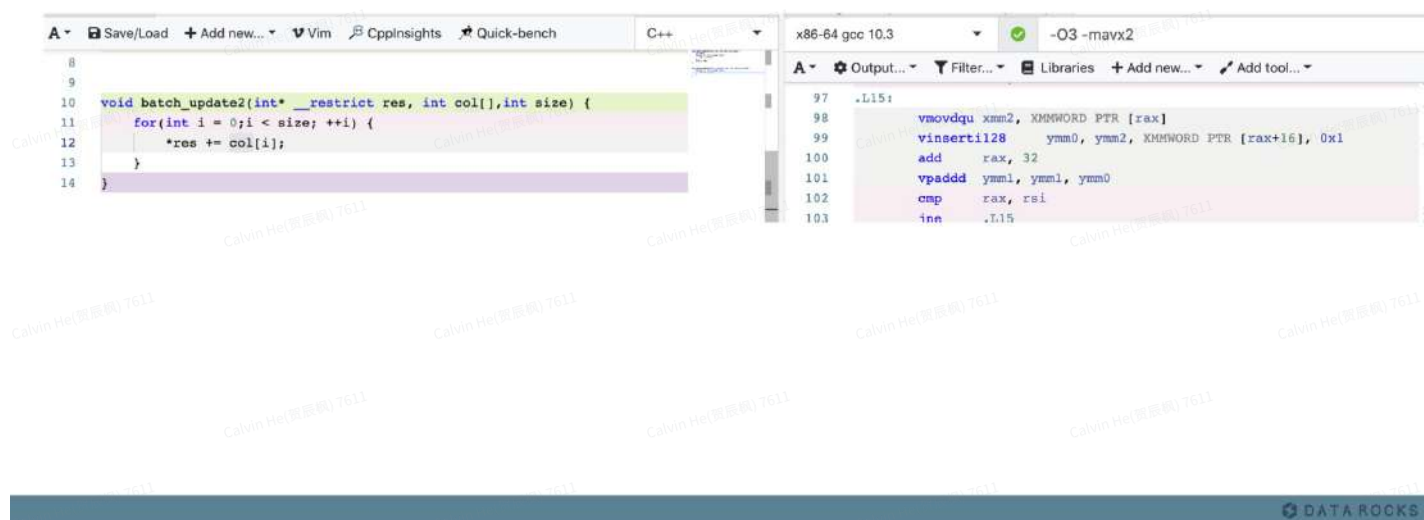


图 5

手动向量化

在进行手动向量化操作之前，需要先熟悉向量化编程的一些常用指令，推荐研发同学浏览 <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html> 官网查询相关细节。

这里以一组简单的代码来直观展示如何手动触发向量化，如图 6 所示，下面一组代码通过手动编译的方式，实现了 6 倍的性能提升。

Vector Intrinsic Examples: HLL Merge

```
for (int i = 0; i < HLL_REGISTERS_COUNT; i++) {  
    _registers.data[i] = std::max(_registers.data[i], other_registers[i]);  
}
```



•Alignment

•Tail Process

•Compatibility

•Only Simple Operation

```
int loop = HLL_REGISTERS_COUNT / 32;  
uint8_t* dst = _registers.data;  
const uint8_t* src = other_registers;  
for (int i = 0; i < loop; i++) {  
    __m256i xa = _mm256_loadu_si256((const __m256i*)dst);  
    __m256i xb = _mm256_loadu_si256((const __m256i*)src);  
    _mm256_storeu_si256((__m256i*)dst, _mm256_max_epu8(xa, xb));  
    src += 32;  
    dst += 32;  
}
```

6X Performance Improvement

DATA ROCKS

图 6

需要注意的是，在手动进行向量化编译时，要明确以下几个关键点，否则会导致出错：

- 内存对齐
- 尾部数据的处理
- 兼容性
- 只能是一些简单的操作

02：数据库如何向量化

StarRocks 从 2020 年 POC 开始至今，两年的落地实践总结了一个重要的认知——**数据库的向量化不仅仅是 CPU 指令的向量化，更是一个巨大的性能优化工程。**

向量化编程对数据库带来的性能提升肉眼可见，但其背后所涉及到的挑战可谓是方方面面，具体包括：

- 数据一直是列式
- 所有算子都需要向量化

- 所有表达式都需要向量化
- 需要尽可能地使用 SIMD 指令
- 需要重新设计内存管理
- 需要重新设计数据结构
- 总体性能提升多少倍，意味着算子和表达式也必须同步提升，不能有短板

数据库向量化的关键点

数据库向量化并非是简单的架构层面的重大变化，涉及到庞大的工程细节，其关键点可以概括为：

1. 数据的组织方式
2. 算子和表达式的向量化
3. SIMD 如何加速 Filter、Agg 和 Join

在这样的背景下，StarRocks 总结了数据库性能优化的 7 大要点，如图 7 所示。下文将结合实例介绍 StarRocks 的落地实践。

Database Vectorized Improvement Methods



1 高性能第三方库

StarRocks 在数据聚合方面使用的是开源的 Parallel Hashmap（图 8），从实际使用体验来看，其在各种场景下的性能都比较出色。除此以外，Fmt、Simdjson、Hyperscan 等也有一些用例。

1 High-Performance Third-Party Lib: Parallel Hashmap

- Parallel Hashmap
- Fmt
- Simdjson
- Hyperscan

Merged [DSDB-3570] Improve HLL aggregate column performance #3260
Changes from all commits File filter Conversations Jump to

be/src/olap/hll.h

namespace doris {
@@ -141,7 +142,7 @@ class HyperLogLog {
private:
HllDataType _type = HLL_DATA_EMPTY;
- std::set<uint64_t> _hash_set;
+ phmap::flat_hash_set<uint64_t> _hash_set;
// This field is much space consuming(HLL_REGISTERS_COUNT)
// it only when it is really needed.
std::vector<uint8_t> _registers;

3X Performance Improvement

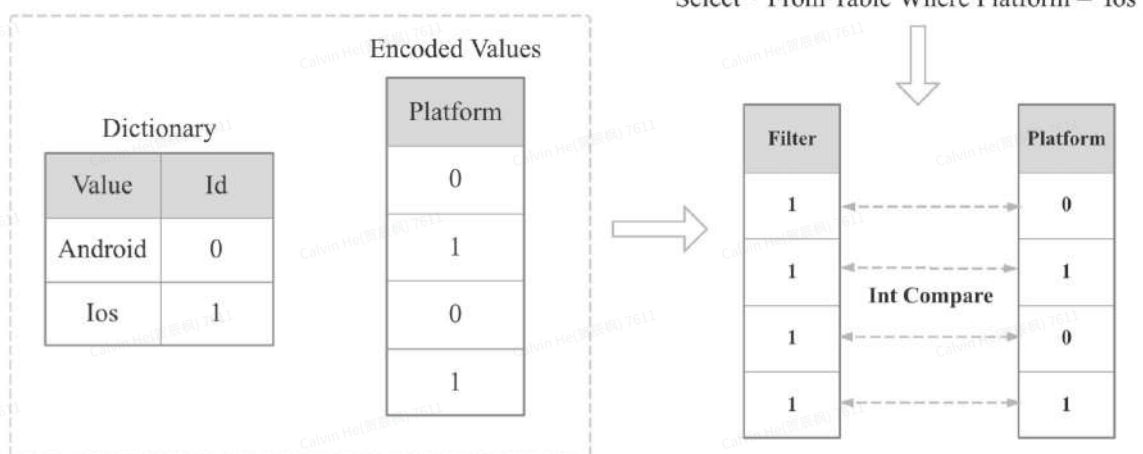
图 8

2 数据结构和算法

数据结构和算法侧的优化思路，是基于编码后的数据集进行操作。在 StarRocks 1.0 版本时代，我们可以通过 Int Compare（图 9）的方式去做计算存储层的优化，这种方式比传统的 String 模式要快很多。

2 Data Structure and Algorithms: Operations On Encoded Data

String Column With Dict Encode



Int Compare is Very Faster Than String

图 9

这种方式存在一定的局限，只能优化"Where"下的数据，于是在 StarRocks 2.0 时代，我们在 MPP 架构下引入了 低基数全局字典（图 10）。这种模式的好处在于，可以将 Scan、Filter、Agg、Sort、Join、String Functions 都囊括在基于编码后的数据集进行操作的优化下，性能提升明显。

2 Data Structure and Algorithms: Operations On Encoded Data

Select Sum(PV) From Table Group By City, Platform

Aggregate Hash Table

Key(Binary)	Sum
AndroidBeijing	40000
AndroidShanghai	30000
IosBeijing	20000
IosShanghai	10000

Aggregate Hash Table With Encoded

Key(Int)	Sum
11	10000
12	20000
21	40000
22	30000

String Encode To TinyInt

Faster Scan
Faster Hash
Faster Equal
Faster Malloc

Platform Dictionary

Value	Id
Android	1
Ios	2

City Dictionary

Value	Id
Beijing	1
Shanghai	2

- Scan
- Filter
- Agg
- Sort
- Join
- String Functions

3X Performance Improvement For Aggregate

图 10

3 自适应策略

Join Runtime Filter Compute (图 11) 比较好理解, 基本原理是通过在Join 的 Probe 端提前过滤掉那些不会命中 Join 的输入数据来大幅减少 Join 中的数据传输和计算, 从而减少整体的执行时间。这个策略的关键在于自适应, 因为数据无法预知, 所以需要尽可能选择有用的 issue, 不做无用的操作。

3 Adaptive Strategy: Join Runtime Filter Compute

```
double selectivity = true_count * 1.0 / chunk_size;
if (selectivity <= 0.5) { // useful filter
    if (selectivity < 0.05) { // very useful filter, could early return
        _selectivity.clear();
        _selectivity.emplace(selectivity, rf_desc);
        chunk->filter(new_selection);
        return;
    }

    // Only choose three most selective runtime filters
    if (_selectivity.size() < 3) {
        _selectivity.emplace(selectivity, rf_desc);
    } else {
```

Prefer The Low Selectivity Filter

DATA ROCKS

图 11

4 SIMD 优化


SIMD 指令前文已经提到很多了, 这里再简单介绍一个字符串函数的例子。如图 12所示, 这段代码主要是用来判断字符串是否符合 Ascii 库标准, 调用 SIMD 指令后实现了 5 倍的性能提升。

4 SIMD Optimization: Improve Ascii Substring

```
char tail_has_error = 0;
for (size_t i = 0; i < len; i++) {
    tail_has_error |= src[i];
}
return !(tail_has_error & 0x80);
```

Validate Ascii String

Ascii Chars From 0x00 To 0x7F



```
__m256i has_error = _mm256_setzero_si256();
if (len >= 32) {
    for (size_t i = 0; i <= len - 32; i += 32) {
        __m256i current_bytes = _mm256_loadu_si256((const __m256i*)
            has_error = _mm256_or_si256(has_error, current_bytes);
    }
}
int error_mask = _mm256_movemask_epi8(has_error);
```

5X Performance Improvement

DATA ROCKS

图 12

5 C++ Low Level 优化

C++ Low Level 的优化是一个很经典的命题，涉及到语言层面的优化业界也有比较多的方法积累，此处不再赘述，具体可参考图 13 所示方法集合，很多优化在代码层面都可以取得成倍的性能优化效果。

5 C++ Low level Optimization

- Inline
- Loop Optimization
- Unrolling
- Resize No Initialize
- Copy To Move
- Std::vector
- Compile-time Computation

图 13

6 内存管理优化

StarRocks 为减轻内存申请释放压力，引入了 HyperLogLog 的内存管理方案，并对其做了二次优化，见图 14。传统的 HyperLogLog 已经能够以极少的内存来统计巨量的数据，而 StarRocks 通过批量分配、内存复用的两个方式，实现了 5 倍的性能提升。

6 Memory Manage: HLL Memory Manage

- Allocate By Chunk
- Reuse Memory

```
+ HyperLogLog::~HyperLogLog() {  
+     if (_registers.data != nullptr) {  
+         ChunkAllocator::instance()->free(_registers);  
+     }  
+ }  
+  
// Convert explicit values to register format, and clear explicit values.  
// NOTE: this function won't modify _type.  
void HyperLogLog::_convert_explicit_to_register() {  
    DCHECK(_type == HLL_DATA_EXPLICIT)  
        << "_type(" << _type << ") should be explicit(" << HLL_DATA_EXPLICIT << ")";  
-    _registers.clear();  
-    _registers.resize(HLL_REGISTERS_COUNT, 0);  
+    DCHECK_EQ(_registers.data, nullptr);  
+    ChunkAllocator::instance()->allocate(HLL_REGISTERS_COUNT, &_registers);  
+    memset(_registers.data, 0, HLL_REGISTERS_COUNT);  
+ }
```

5X Performance Improvement

DATA ROCKS

图 14

7 CPU Cache 优化

每一个做性能优化的人都应该去关注 CPU Cache，因为 Cache Miss 所导致的延迟惩罚是成指数级提升的。另一方面，CPU 的性能瓶颈是动态变化的，有可能优化了 A，但瓶颈又出现在了 B 上。因此，CPU Cache 的优化非常重要，这里总结有以下几个优化方法：

1. 局部的优化（空间和时间）
2. 对齐数据和代码
3. 减少内存访问范围
4. Block 策略
5. Prefetch 优化

03: StarRocks 向量化工程的总结与思考

前文大体介绍了向量化编程的一些基本概念和 StarRocks 在实践中的具体落地经验，最后再跟读者朋友分享一些关于向量化工程的总结与思考。

其一，无论是 CPU 架构还是数据库向量化，其底层原理都是相似的。比如，CPU Frontend 负责指令的取码、编解码，Backend 负责执行。相类似的，StarRocks 中的 Frontend(Query Plan)负责做计划，Backend 负责做执行。

其二，打造一个高性能的数据库，需要的不仅是底层架构，还要保证持续的工程细节优化。比如业界知名的 ClickHouse 数据库，可能在架构设计层面上没有优秀到远超同类，它完全是为了性能，自下而上地构建自己的架构。

其三，向量化和查询编译的融合。为了提升 CPU 的性能，向量化和查询编译是两条道路，但二者之间其实是正交关系，可以在查询编译的同时，生成向量化代码。具体来说，复杂的表达式、UDF, UDAF, UDTF、Sort, Aggregate 以及 GPU 等都可以尝试。

其四，引入 GPU、FPGA 等新硬件。CPU 的性能开发当前并未到达极限，但需要投入的精力已经远非几年前可比。在这样的条件下，引入新的硬件到数据库领域，或许可以带来爆发式的性能增长。

其五，挑战不可能。StarRocks 从诞生之初到现在的蓬勃发展，包括我们团队日程攻坚的向量化编程的相关工作，都是在践行挑战不可能的这种精神。我们需要去打破思维的定式，仰望星空，脚踏实地，去做创新的工作。

作者



康凯森 | StarRocks PMC、StarRocks 查询方向负责人